Accepted and to appear in *IEEE Trans. on Parallel and Distributed Systems*.

On Effective Execution of Non-Uniform DOACROSS Loops *

Ding-Kai Chen

dkchen@mti.sgi.com

Silicon Graphics

Computer Systems

Pen-Chung Yew

yew@cs.umn.edu

Dept. of Computer Science

University of Minnesota

February 28, 1995

Abstract

It is extremely difficult to parallelize *DOACROSS* loops with non-uniform loop-carried dependences. In this paper, we present a static scheduling scheme with an accompanying synchronization strategy that can execute such *DOACROSS* loops effectively and efficiently. Our approach uses one of the parallelization techniques called *Dependence Uniformization*, which finds a small set of uniform dependence vectors to cover all possible non-uniform dependences in a *DOACROSS* loop. It differs from the previous schemes in that we demonstrate a better way to select the uniform dependence vectors. When used with the *Static Strip Scheduling* scheme, the proposed uniform dependence vectors set allows us to enforce dependences with more locality, which reduces the requirement of explicit synchronization considerably while retaining most of the parallelism. This paper describes the uniform dependence vectors selection strategy and the static strip scheduling scheme. The performance analysis and examples are also presented.

*This work was supported in part by the National Science Foundation under Grant Nos. NSF MIP89-20891, and NSF MIP93-07910, and the U.S. Department of Energy under grant No. DE-FG02-85ER25001.

Contents

1 Introduction

Loop-level parallelism is very important in scientific codes. Many architectures and optimization compilers are aimed at efficient execution of parallel loops. One kind of DO loops, the *DOALL* loops, does not have loop-carried dependences, and it is relatively easy to obtain the desired speedup. On the other hand, it is much harder to parallelize *DOACROSS* loops [18, 19, 10, 11], which contain loop-carried dependences.[1] They require the consideration of not only the scheduling and load-balancing issues but also the synchronization issues, which are usually much more difficult.

There are several ways to deal with a *DOACROSS* loop. If possible, we might break the dependences and transform the *DOACROSS* loop into a *DOALL* loop. Many techniques, such as induction variable recognition, variable renaming, loop alignment, scalar expansion, and loop distribution can be used to break such dependences [18, 29, 17]. If this is not possible, we can still execute the loop in parallel, provided proper synchronization is added to enforce loop-carried dependences. The need for synchronization can be determined either at compile-time [15, 25] for uniform (i.e., constant distance) dependences, or at run-time [30, 6] for complicated non-uniform dependences. If all these techniques fail, the *DOACROSS* loop must be executed serially which, according to Amdahl's law [2], could severely degrade the performance [7].

The most common type of loop-carried dependences in engineering and scientific programs arises from references to the same array element in different loop iterations. In many cases, the synchronization overhead can be excessive to execute the loop in parallel, and the loop should remain serial. A recent empirical study showed that, before compiler optimizations such as forward substitution, induction variable elimination, and constant folding are applied, 66% of the array references have linear or partially linear subscript expressions [23]. It also showed that approximately 45% of the reference pairs with linear or partially linear subscript expressions have *coupled subscripts*; that is, the same loop index variable appears in more than one dimension. They are one of the major causes of non-uniform data dependences. Another major cause of non-uniform data dependences arises from array references in different loop nesting levels of a non-perfectly nested loop. In [27], an approach is proposed to handle non-uniform dependences in DOACROSS loops. However, it does not achieve high parallelization efficiency in many cases.

In this paper we propose new scheduling and synchronization strategies that can greatly improve the speedup and reduce the synchronization overhead of such techniques. We organize our presentation as follows. Section 2 introduces the

[1] The definition of DOACROSS loops here includes all the loops with loop-carried dependences. The definition is broader than the traditional DOACROSS loop definition, which includes only the loops with dependence cycles.

program model and some background needed for later discussion. Our proposed uniformization scheme and the *Static Strip Scheduling* scheme are explained in Section 3, and the performance is analyzed in detail in Section 4. We present an example in Section 5. Special cases and further optimization techniques are discussed in Section 6. Section 7 describes related work, and the concluding remarks are given in Section 8.

2 Background

2.1 Program Model

The program model for the loop nest considered in this paper has the following form

```
DO I1=L1,U1,1
        DO I2=L2(I1),U2(I1),1
                H2a
                DO I3=L3(I1,I2),U3(I1,I2),1
                        H3a
                        ⋮
                        DO Ilc=Llc(I1,I2,...,Ilc-1), Ulc(I1,I2,...,Ilc-1),1
                                H
                        END DO
                        ⋮
                        H3b
                END DO
                H2b
        END DO
END DO
```

Figure 1: The program model.

where $H_{2a}, H_{3a}, \ldots, H_{3b}, H_{2b}$, and H are sequences of simple statements such as assignments, subroutine calls, or compound statements such as IF and DO statements. The scheme is very general in that it can handle non-perfect multiply-nested loops with loop limits expressed in functions of outer loop index variables. The only requirement is that the two outermost loops are perfectly nested and they are the loops to be executed in parallel.[2] Each of the lower and upper loop limits, L_i and U_i, is an affine function of $(I_1, \ldots, I_{i-1})$ (all coefficients are integer constants). We use loop $l_c \geq 2$ to denote the innermost

[2]The same strategy can be easily generalized to more than 2 loop levels, as shown in Section 6, provided that the parallel loops are perfectly nested.

common loop of the statements involved in a dependence relation.

2.2 Data Dependences and Data Synchronization

A *data dependence* occurs when a data object x is accessed by two statements S_p and S_q, and there exists an execution path between S_p and S_q. Also, at least one of the accesses is a write access. In this paper, we are only interested in the dependences caused by references to the array elements. The subscript functions of the array elements are assumed to be affine functions of the loop indices of the enclosing loops. We use $S_k(\vec{I})$, where $\vec{I} = (i_1, i_2, \ldots, i_k)$ is a *loop index vector*, to denote a particular instance of statement S_k in iteration $(I_1 = i_1, I_2 = i_2, \ldots, I_k = i_k)$. We also assume that there is no exit-loop statement in the loop nest.

Definition 1 *If a dependence occurs such that $S_q(\vec{J})$ has to wait for the completion of $S_p(\vec{I})$, the dependence is said to have the source $S_p(\vec{I})$ and the sink $S_q(\vec{J})$.*

Loop-carried dependences can be described by the *dependence distance vectors*:

Definition 2 *If a dependence has the source $S_p(\vec{I})$ and the sink $S_q(\vec{J})$, its dependence distance vector, or simply the dependence vector, is $\vec{J} - \vec{I} = (j_1 - i_1, j_2 - i_2, \ldots, j_{l_c} - i_{l_c})$, where l_c is the innermost common loop to both S_p and S_q.*

The dependence can be further categorized as *uniform* and *non-uniform*.

Definition 3 *A dependence is uniform if the dependence distance vector is a constant vector. That is, the distance is the same for all iterations. Otherwise, the dependence is non-uniform.*

Many techniques have been proposed to transform loops with loop-carried dependences to vector loops or *DOALL* loops [29, 3, 1]. To parallelize remaining loops with loop-carried dependences, proper data synchronization instructions have to be generated by the compiler to preserve the dependences. Synchronization for a uniform dependence is relatively easy to generate [15, 25] because the distance between the source and the sink statement is fixed. However, for a non-uniform dependence, it is very difficult because each iteration has varying dependence distances. In most cases, loops with non-uniform dependences must be executed serially. Consider the loop in Figure 2. The dependence vectors vary in each iteration of the J-loop. Generating synchronization instructions for each J-loop iteration becomes very inefficient if not impossible.

```
         DO I=1,1000
            DO J=1,I
               DO K=J,I
                  ⋮
Sp:               ... =A(I+2K+5,4K-J)
               ENDDO
               ⋮
Sq:            A(I-J,I+J)= ...
               ⋮
            ENDDO
         ENDDO
```

Dep. source	Dep. sink	Dep. distance
...		
S_p(16,1,12)	S_q(46,1)	(30,0)
S_p(16,1,13)	S_q(49,2)	(33,1)
S_p(16,1,14)	S_q(52,3)	(36,2)
S_p(16,1,15)	S_q(55,4)	(39,3)
S_p(16,1,16)	S_q(58,5)	(42,4)
S_p(16,3,13)	S_q(48,1)	(32,-2)
S_p(16,3,14)	S_q(51,2)	(35,-1)
S_p(16,3,15)	S_q(54,3)	(38,0)
S_p(16,3,16)	S_q(57,4)	(41,1)
...		

Figure 2: A loop with non-uniform dependence distance vectors.

2.3 Dependence Analysis

Although the dependence distance vectors shown in Figure 2 are not uniform, they can be described in a fairly compact mathematical form. Suppose we have two accesses to an s-dimensional array A, $A(f_{p_1}(\vec{I_p}), f_{p_2}(\vec{I_p}), \ldots, f_{p_s}(\vec{I_p}))$ at statement S_p, and $A(f_{q_1}(\vec{I_q}), f_{q_2}(\vec{I_q}), \ldots, f_{q_s}(\vec{I_q}))$ at statement S_q, where $\vec{I_p} = (I_1, \ldots, I_{l_p})$, $\vec{I_q} = (I_1, \ldots, I_{l_q})$. l_p and l_q are the number of loops enclosing S_p and S_q, respectively, with $l_p \geq l_c$ and $l_q \geq l_c$. Each $f_{p_i}, f_{q_i}, i = 1, \ldots, s$, is an integer affine function of loop indices. To identify the source and the sink iterations that need to synchronize with each other, one has to solve the following simultaneous linear equations and obtain integer solutions for each component of $\vec{I}(= \vec{I_p})$ and $\vec{J}(= \vec{I_q})$

$$\left\{ \begin{array}{ccc} f_{p_1}(i_1, i_2, \ldots, i_{l_p}) & = & f_{q_1}(j_1, j_2, \ldots, j_{l_q}) \\ f_{p_2}(i_1, i_2, \ldots, i_{l_p}) & = & f_{q_2}(j_1, j_2, \ldots, j_{l_q}) \\ & \vdots & \\ f_{p_s}(i_1, i_2, \ldots, i_{l_p}) & = & f_{q_s}(j_1, j_2, \ldots, j_{l_q}) \end{array} \right\} \tag{1}$$

under the constraints specified by the loop limits of the $l_p + l_q$ enclosing loops

$$\left\{ \begin{array}{lcl} L_1 & \leq & i_1 \leq U_1 \\ L_2(i_1) & \leq & i_2 \leq U_2(i_1) \\ & \vdots & \\ L_{l_c}(i_1, \ldots, i_{l_c - 1}) & \leq & i_{l_c} \leq U_{l_c}(i_1, \ldots, i_{l_c - 1}) \\ & \vdots & \\ L_{l_p}(i_1, \ldots, i_{l_p - 1}) & \leq & i_{l_p} \leq U_{l_p}(i_1, \ldots, i_{l_p - 1}) \\ \\ and & & \\ \\ L_1 & \leq & j_1 \leq U_1 \\ L_2(j_1) & \leq & j_2 \leq U_2(j_1) \\ & \vdots & \\ L_{l_c}(j_1, \ldots, j_{l_c - 1}) & \leq & j_{l_c} \leq U_{l_c}(j_1, \ldots, j_{l_c - 1}) \\ & \vdots & \\ L_{l_q}(j_1, \ldots, j_{l_q - 1}) & \leq & j_{l_q} \leq U_{l_q}(j_1, \ldots, j_{l_q - 1}) \end{array} \right\} \tag{2}$$

In our example, $s = 2$ and $l_c = 2$ and there is a probable data dependence between $S_p(i_1, i_2, i_3)$ and $S_q(j_1, j_2)$. The equations to be solved are:

$$\left\{ \begin{array}{lcl} i_1 + 2i_3 + 5 & = & j_1 - j_2 \\ -i_2 + 4i_3 & = & j_1 + j_2. \end{array} \right\} \tag{3}$$

We can use the algorithm presented in [3] to determine their integer solutions. If there is no integer solution, no synchronization is needed for this dependence. If there is exactly one solution, only the two iterations involved, $\vec{I_p}$ and $\vec{I_q}$, need to synchronize their execution. Otherwise there is more than one solution, which can be expressed as

$$(i_1, i_2, \ldots, i_{l_p}, j_1, j_2, \ldots, j_{l_q}) = (g_{i_1}(T), g_{i_2}(T), \ldots, g_{i_{l_p}}(T), g_{j_1}(T), g_{j_2}(T), \ldots, g_{j_{l_q}}(T)) \tag{4}$$

where $T = (t_1, \ldots, t_{l_p + l_q - m})$ and $t_i, 1 \leq i \leq l_p + l_q - m$ are free variables if m out of the s equations in Eq. (1) are linearly independent. Each g in Eq. (4) is an integer linear function of $t_i, i = 1, \ldots, l_p + l_q - m$. Applying this to our example, the general form of solutions to Eq. (3) is

$$(i_1, i_2, i_3, j_1, j_2) = (t_1, t_1 - 2t_2 + 5, t_3, t_2 + 3t_3, -t_1 + t_2 + t_3 - 5)$$

where t_1, t_2, and t_3 are free variables. The constraints from the loop limits are:

$$\left\{ \begin{array}{lllll} 1 & \le & t_1 & \le & 1000 \\ 1 & \le & t_1 - 2t_2 + 5 & \le & t_1 \\ t_1 - 2t_2 + 5 & \le & t_3 & \le & t_1 \\ 1 & \le & t_2 + 3t_3 & \le & 1000 \\ 1 & \le & -t_1 + t_2 + t_3 - 5 & \le & t_2 + 3t_3 \end{array} \right\} \tag{5}$$

The inequalities of Eq. (2) form a set of half spaces and the intersection of them is a *convex polytope* [22]. (See Theorem 4 in Appendix. It is called the *Dependence Convex Hull* or *DCH*. Because loop l_c is the innermost common loop to S_p and S_q, the general form of the dependence distance vector is

$$(d_1(T), d_2(T), \ldots, d_{l_c}(T)) = (g_{j_1}(T) - g_{i_1}(T), g_{j_2}(T) - g_{j_2}(T), \ldots, g_{j_{l_c}}(T) - g_{i_{l_c}}(T)). \tag{6}$$

The corresponding dependence vector in our example can be summarized as

$$(d_1, d_2) = (j_1 - i_1, j_2 - i_2) = (-t_1 + t_2 + 3t_3, -2t_1 + 3t_2 + t_3 - 10) \tag{7}$$

or, in matrix form

$$\begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} j_1 - i_1 \\ j_2 - i_2 \end{bmatrix} = \begin{bmatrix} -1 & 1 & 3 \\ -2 & 3 & 1 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} + \begin{bmatrix} 0 \\ -10 \end{bmatrix} \tag{8}$$

Note that the above representation of a dependence distance vector includes those with a negative leading non-zero component. They correspond to the reversal of a valid dependence. It means that for some value of T, S_q is the dependence source and S_p is the dependence sink, instead of the other way around, as is assumed when we construct the Eq. (6). However, our scheme, proposed in the next section, considers both choices of source and sink statement; therefore, is not affected by this problem.

3 Parallelizing Non-uniform Dependence Loops

For loops with non-uniform dependences, we can *uniformize* the dependences [27] to simplify required synchronization. The uniformization process is illustrated as in Figure 3. In a 2-dimensional iteration space, if the source and the sink iteration

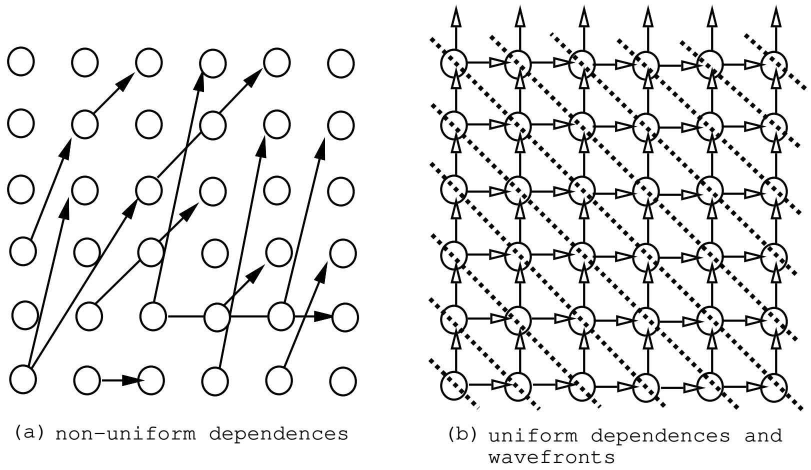

Figure 3: An example of non-uniform dependences and parallelization using the wavefront method.

of every non-uniform dependence (as in Figure 3a) can be connected through a combination of a set of uniform dependence vectors (as $\{(0,1),(1,0)\}$ in Figure 3b), enforcing that set of vectors on *every* iteration can preserve all non-uniform dependences. Basically, there are two major steps in this process:

1. Find a set of uniform dependence vectors that can *cover* all probable non-uniform dependences in the loop nest.

2. Devise a loop scheduling and synchronization strategy to execute the loop nest in parallel.

There can be many ways to parallelize loops as described above. Our goals, however, is to have maximal parallelism. Otherwise, a uniformization scheme can essentially serialize a loop with a small set of uniform dependences.

3.1 Uniformization Examples

In this section, we present a few examples of the sets of uniform dependence vectors, called *Basic Dependence Vector Sets* or *BDVSs*, which can cover all the non-uniform dependences in Figure 2. There are infinitely many choices for a *BDVS*. However, to make it practical,

- the size of a *BDVS* must be as small as possible to minimize synchronization overhead; and
- it must provide a good speedup. That is, it must be chosen in the context of some good scheduling scheme given the number of processors available.

Because an optimal schedule is expensive to compute and we want to compare the *BDVS*s independently of the number of processors, we use instead the *dependence cone size* as the measure of choosing a good *BDVS*.

Definition 4 (Dependence cone and dependence cone size [9]) *For a dependence vector set* $V = \{\vec{v_1}, \vec{v_2}, \ldots, \vec{v_m}\}$*, the dependence cone denoted* $C(V)$ *is defined as the set*

$$C(V) = \{\vec{v} \in \mathbf{R^n} : \vec{v} = \lambda_1\vec{v_1} + \lambda_2\vec{v_2} + \ldots + \lambda_m\vec{v_m}, \lambda_1, \ldots, \lambda_m \geq 0\}$$

and the dependence cone size is defined as, assuming each $\vec{v} = (v_1, v_2, \ldots, v_n)$*:*

$$\int_{\vec{v} \in C(V), v_1^2+v_2^2+\ldots+v_n^2=1} d\vec{v}.$$

In other words, the dependence cone size is defined as the area of the intersection of $v_1^2 + v_2^2 + \ldots + v_n^2 = 1$ *with dependence cone* $C(V)$*.*

Basically, the dependence cone of a dependence vector set V is the set of all possible non-negative linear combinations of the vectors in V (i.e., the shaded area in Figure 4). In the 2-dimensional case, the dependence cone size is in fact proportional to the "angle" expanded by the cone (see Figure 4). The significance of the dependence cone size can be explained by using the *wavefront method* [16, 13] as shown in Figure 3b where all of the iterations on each "wavefront" can be executed in parallel. More formally, it is a linear mapping

$$\pi(i_1, \ldots, i_n) = \alpha_1 i_1 + \ldots + \alpha_n i_n = (\alpha_1, \ldots, \alpha_n) \cdot (i_1, \ldots, i_n) = \vec{\Pi} \cdot (i_1, \ldots, i_n) \qquad (9)$$

which maps an iteration $(i_1, \ldots, i_n)$ to $\pi(i_1, \ldots, i_n)$. The mapping ensures that

$$((j_1, \ldots, j_n)\ depends\ on\ (i_1, \ldots, i_n)) \Rightarrow (\pi(j_1, \ldots, j_n) > \pi(i_1, \ldots, i_n)) \Rightarrow \vec{\Pi} \cdot (j_1 - i_1, \ldots, j_n - i_n) > 0 \qquad (10)$$

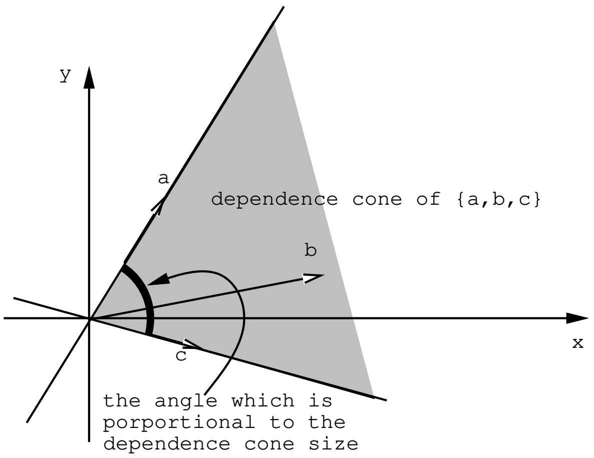

Figure 4: 2-dimensional dependence cone and dependence cone size.

to preserve dependences. The set of iterations $(i_1, \ldots, i_n)$ with the same value of $\pi(i_1, \ldots, i_n)$ is said to be on the same wavefront. An important decision when applying the wavefront method is to determine the angle for the wavefronts. It can determine the total number of wavefronts needed to execute the loop. This is equivalent to determining the vector $\vec{\Pi} = (\alpha_1, \ldots, \alpha_n)$ in Eq. (9). The constraint on the choice of $\vec{\Pi}$ is specified in Eq. (10) that $\vec{\Pi} \cdot \vec{D} > 0$ for every dependence vector $\vec{D}$. For example, Figure 5 shows two dependence cones (dark-shaded area) in the 2-D dependence vector space. The feasible solution space for $\vec{\Pi}$ with the constraints specified in Eq. (10), which is equivalent to $\vec{\Pi} \cdot \vec{D} > 0$ for all dependence vector $\vec{D}$, is the light-shaded area in Figure 5. In general, a larger dependence cone implies a smaller solution space for $\vec{\Pi}$ as shown in Figure 5a. The following theorem further shows the relation between the dependence cone size and the total execution time when a dependence cone is contained in another dependence cone.

Theorem 1 *[9] Let* $\vec{v_1}, \vec{v_2}, \ldots, \vec{v_m}$ *be the vectors in* BDVS_1 *and* $\vec{u_1}, \vec{u_2}, \ldots, \vec{u_k}$ *be the vectors in* BDVS_2*. If* $\forall i = 1, \ldots, m$ ($\vec{v_i} = \sum_{j=1}^{k} \lambda_{i,j} \vec{u_j}$ *where* $\lambda_{i,j} \geq 0$), *and if* $\forall i = 1, \ldots, m$ ($\sum_{j=1}^{k} \lambda_{i,j} \geq 1$), *then* $C(\text{BDVS}_1) \subseteq C(\text{BDVS}_2)$*, where* $C(\text{BDVS})$ *is the dependence cone for* BDVS*. Also,* $t(\text{BDVS}_1) \leq t(\text{BDVS}_2)$ *where* $t(\text{BDVS})$ *is the execution time of the loop using the corresponding optimal linear schedule of* BDVS.

Proof: See [9] for details. □

That is, if the dependence cone for $BDVS_1$ is contained in $BDVS_2$, then the solution space for $\vec{\Pi}_2$ of $BDVS_2$ must be contained in the solution space for $\vec{\Pi}_1$ of $BDVS_1$. Therefore, the best choice of the vector $\vec{\Pi}_2$ found for $BDVS_2$ can

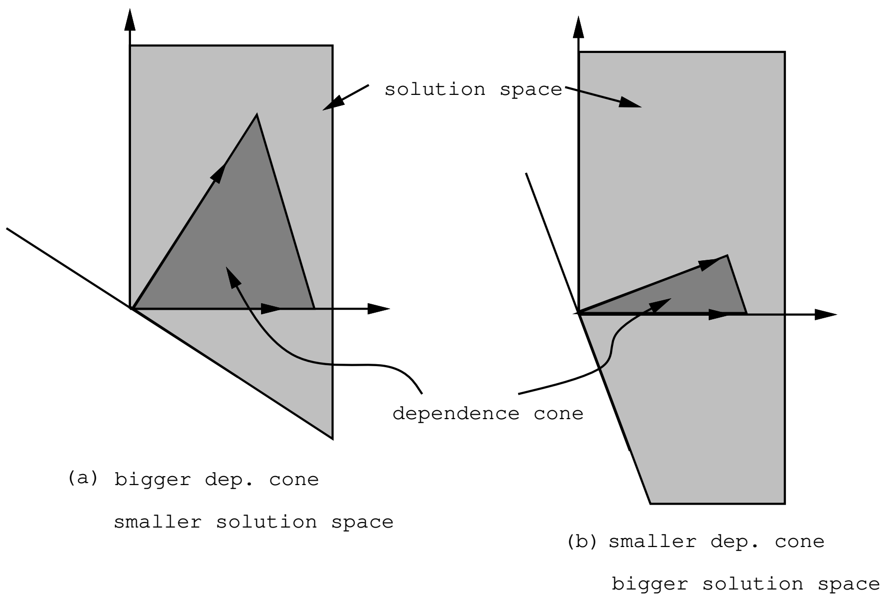

Figure 5: Dependence cone and solution space of linear schedule vector.

always be used for $BDVS_1$, and the execution time for $BDVS_1$ should be no larger than the execution time for $BDVS_2$. To summarize the discussion so far, we prefer the *BDVS* to have a smaller dependence cone size, which will have a shorter parallel execution time using, for example, the wavefront method.

We now look at a few *BDVSs* for the non-uniform dependences in Eq. (7). Each dependence must be able to be decomposed as a non-negative linear combination of vectors in each *BDVS*. Some of them are illustrated in Figure 6.

Example 1 (Naive Decomposition) BDVS=$\{(1,0),(0,1),(0,-1)\}$ *(see Figure 6a).*

It is easy to see that any valid dependence vector $\vec{D} = (x, y)$ in a doubly nested loop can be decomposed as $\vec{D} = x \times (1,0) + y \times (0,1)$ if $y \geq 0$ or $\vec{D} = x \times (1,0) + (-y) \times (0,-1)$ if $y < 0$. Note that we must have $x \geq 0$ for all valid dependence vectors because x corresponds to the outermost loop level. The problem with this decomposition is that $(0,1)$ and $(0,-1)$ are not compatible with each other. That is, if we have (x,y) executed before $(x, y+1)$ because of $(0,1)$, then we cannot have $(0,-1)$ because it will mean that we also can have $(x, y+1)$ executed before (x,y).

Example 2 (Basic Method I in [9]) BDVS=$\{(1,0),(0,1),(22,-323)\}$.

This *BDVS* is very similar to the previous one except for the last vector $(22,-323)$. It is derived from lower bounds of each component of all the non-uniform dependence vectors. Such lower bounds can be obtained with a linear programming

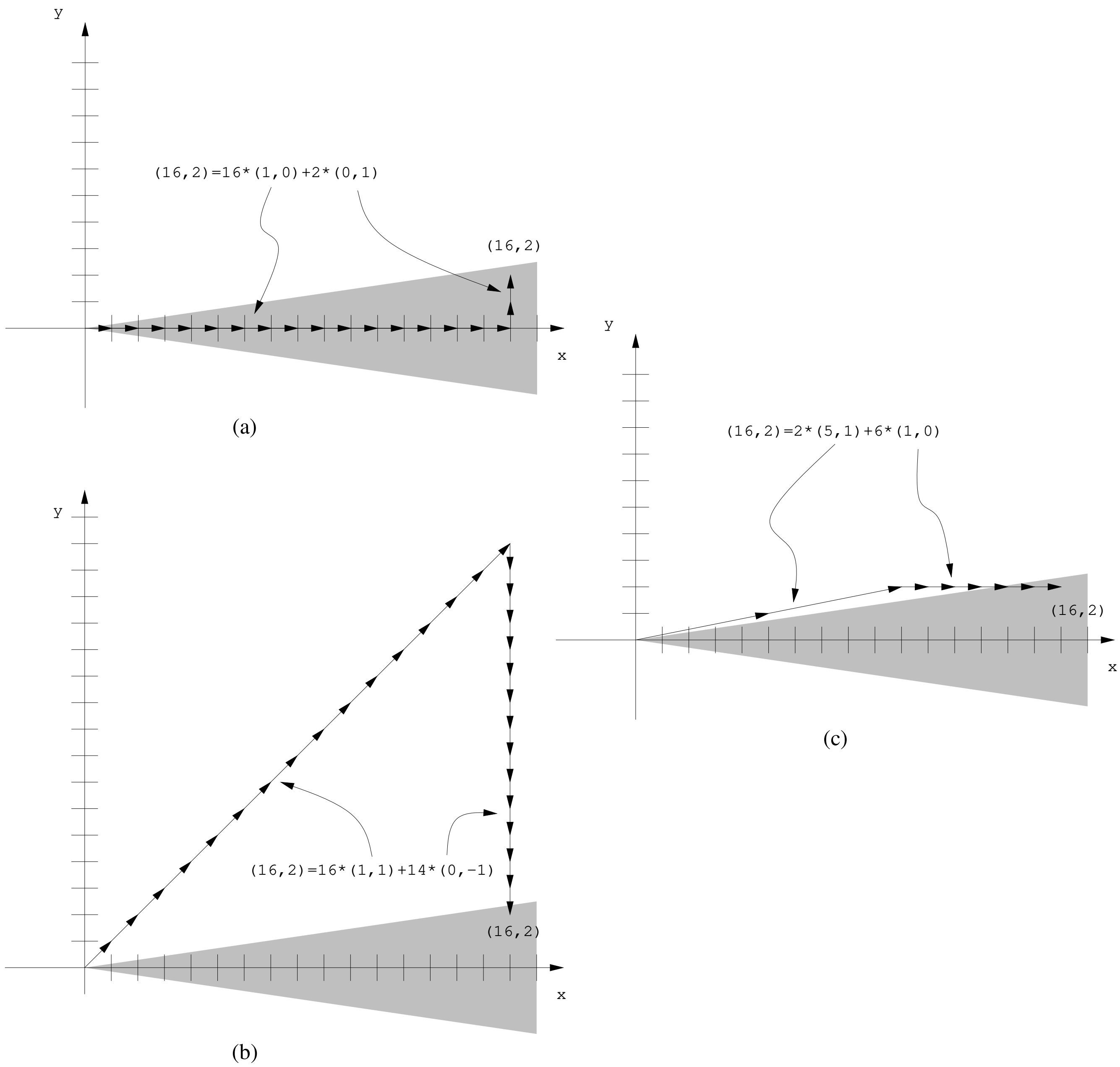

Figure 6: Illustration of (a) Naive decomposition (Example 1), (b) S-test (Example 4), (c) Proposed decomposition (Example 5). Shaded areas are possible sink iterations of non-uniform dependences when the origin is the current iteration.

technique to optimize the problem, $min_{T \in DCH}$ $\{d_i(T)\}$ for $i = 1, 2$, where *DCH* is the dependence convex hull discussed in Section 2.3 [9]. Let x_{min} and y_{min} be the minimum of d_1, d_2, respectively. Then any valid dependence vector $\vec{D} = (x, y)$ in the doubly nested loop can be decomposed as $\vec{D} = (x - \lceil x_{min} \rceil) \times (1, 0) + (y - \lceil y_{min} \rceil) \times (0, 1) + 1 \times (\lceil x_{min} \rceil, \lceil y_{min} \rceil)$. The compatibility problem of the previous example does not exist in this *BDVS*. In our example, $x_{min} = 22$ and $y_{min} = -323.33$ and the size of the dependence cone is $\frac{\pi}{2} + \arctan(\frac{323}{22}) = 3.06$, which is close to the upper limit, π. In fact, this is not entirely correct because the *DCH* so far does not consider the direction of the dependences. As a result, the dependences represented by Eq. (8) include some of the inverted real dependences. This problem can be eliminated by enumerating all possible direction vectors and combining them with the *DCH* to obtain the ultimate minimal values.

Example 3 (Basic Method II in [9]) BDVS=$\{(-1, -2), (1, 3), (3, 1), (0, -10)\}$.

The validity argument of this *BDVS* follows from Eq. (8). Because t_1, t_2, and t_3 are all positive as indicated in Eq. (5), any valid dependence vector $\vec{D} = (x, y)$ in a doubly nested loop can be decomposed as $\vec{D} = t_1 \times (-1, -2) + t_2 \times (1, 3) + t_3 \times (3, 1) + (0, -10)$ according to Eq. (8). The problem with this *BDVS* is that the number of free variables and the dependence cone size can potentially be large. In addition, similar to the previous example, some of the positive linear combinations of the members of this *BDVS* will include the reversal of some of the real dependence vectors. It will cause the sink statements of some dependences to be executed before the source statements. What is worse than the previous example is that this *BDVS* is independent of the *DCH*. Therefore the direction vectors will not help. Another problem with this *BDVS* is that it works for only one dependence. Different *BDVSs* have to be found for different dependences in the loop. Hence, the growth of the *BDVS*'s cardinality and the synchronization overhead can be substantial. In short, although the construction of this *BDVS* is straightforward, it works only if there are only a few dependences in the loop and it is clear at the compile time that one statement is always the source and the other one statement is always the sink.

Example 4 (S-test in [27]) BDVS=$\{(1, 1), (0, -1)\}$ *(see Figure 6b).*

This *BDVS* consists of only two uniform dependence vectors $(1, \lceil sl_{max} \rceil)$ and $(0, -1)$ where sl_{max} is an upper bound of $\left\{\frac{d_2(T)}{d_1(T)}, \forall T \in DCH\right\}$. For any valid dependence vector $\vec{D} = (d_1, d_2)$, $\frac{d_2}{d_1} \leq sl_{max}$ and $d_1 \geq 0$. Therefore, $d_2 \leq d_1 \times sl_{max} \leq d_1 \times \lceil sl_{max} \rceil$. Because both d_1 and d_2 are integers, $d_2 = d_1 \times \lceil sl_{max} \rceil - r$, $r \in \mathbf{Z}^+ \cup \{0\}$, where $\mathbf{Z}^+$ is the set of positive integers. In other words, $\vec{d} = (d_1, d_2) = (d_1, d_1 \times \lceil sl_{max} \rceil - r) = d_1 \times (1, \lceil sl_{max} \rceil) + r \times (0, -1)$. In our example, the maximal value of $\frac{d_2(T)}{d_1(T)} = 0.194$ which results in a *BDVS* with a dependence cone size of $\frac{3\pi}{4} = 2.36$.

Example 5 **(Proposed** ***BDVS*** **[8])** BDVS=$\{(1,0),(5,1),(2,-1)\}$ *(see Figure 6c).*

The *BDVS* in this example has the smallest dependence cone size, 0.66, in comparison with the previous ones. This is the kind of *BDVS* generated by our proposed scheme. There are several properties of such *BDVSs*:

- small dependence cone size
- no violation of the dependence direction vectors as in Examples 2 and 3
- small cardinality (between n and $2n-1$ for n dimensional iteration space)

More importantly, it works very well with our *Static Stripe Scheduling* scheme to obtain good speedup and to minimize required explicit synchronizations. In the next section, we introduce the theorems which provide the foundation of our scheme.

3.2 Effective Dependence Uniformization

Given a set of dependence vectors $V = \{\vec{v_1}, \vec{v_2}, \ldots, \vec{v_m}\}$, where each $\vec{v_i} = (v_{i,1}, \ldots, v_{i,n})$ is a dependence vector in the n-dimensional iteration space, we can find a *BDVS* with cardinality less than or equal to $2n-1$. Let ub_k^j and lb_k^j be such that

$$ub_k^j \geq \frac{v_{i,j}}{v_{i,k}} \geq lb_k^j, \forall i \in \{1, \ldots, m\}$$

for each $j, k \in \{1, \ldots, n\}$ where k is some appropriately selected dimension, of which we want to represent some j dimension. That is, $\forall i \in \{1, \ldots, m\}$, ub_k^j and lb_k^j are upper and lower bounds of $\frac{v_{i,j}}{v_{i,k}}$. Our uniformization scheme is based on the results of the next two theorems which show how to represent the jth component of a dependence vector with its kth component and the bounds of their ratios. That is, when $v_{i,k} \geq 0$, $v_{i,j} = \lfloor lb_k^j \rfloor \times v_{i,k} + r, r \in \mathbf{Z}^+ \cup \{0\}$ (Theorem 2); and when $v_{i,k} \leq 0$, $v_{i,j} = \lfloor -ub_k^j \rfloor \times (-v_{i,k}) + r, r \in \mathbf{Z}^+ \cup \{0\}$ (Theorem 3). Note that both $\lfloor lb_k^j \rfloor$ and $\lfloor -ub_k^j \rfloor$ are independent of the vector $\vec{v_i}$. The following lemma is useful for the later theorems.

Lemma 1 *Let $a, b, c \in \mathbf{R}$. If $a \times d \geq b \geq c \times d$ and d is a positive integer, then $\lceil a \rceil \times d \geq \lceil b \rceil \geq b \geq \lfloor b \rfloor \geq \lfloor c \rfloor \times d$.*

Proof: We only need to show that $\lceil a \rceil \times d \geq \lceil b \rceil$ and $\lfloor b \rfloor \geq \lfloor c \rfloor \times d$. Let $a = \lceil a \rceil - a_r$ where $0 \leq a_r < 1$. $b \leq a \times d \Rightarrow \lceil b \rceil \leq \lceil a \times d \rceil = \lceil \lceil a \rceil \times d - a_r \times d \rceil \leq \lceil \lceil a \rceil \times d \rceil = \lceil a \rceil \times d$. Similarly, let $c = \lfloor c \rfloor + c_r$ where $0 \leq c_r < 1$.

$b \geq c \times d \Rightarrow \lfloor b \rfloor \geq \lfloor c \times d \rfloor = \lfloor \lfloor c \rfloor \times d + c_r \times d \rfloor \geq \lfloor \lfloor c \rfloor \times d \rfloor = \lfloor c \rfloor \times d.$ □

The next two theorems are more general so that they can be extended to a higher dimensional case by varying the positive integer k_d (see Section 6) and they can be used to explain other uniformization schemes, such as that in Example 4.

Theorem 2 *For a dependence vector $\vec{v}$, if $v_k \geq 0$, then*

$$\lceil \frac{ub_k^j}{k_d} \rceil \times v_k \geq \lceil \frac{v_j}{k_d} \rceil \geq \frac{v_j}{k_d} \geq \lfloor \frac{v_j}{k_d} \rfloor \geq \lfloor \frac{lb_k^j}{k_d} \rfloor \times v_k$$

where k_d is a positive integer.

Proof: For every dependence vector $\vec{v}$, $ub_k^j \geq \frac{v_j}{v_k} \geq lb_k^j$ and $v_k \geq 0 \Rightarrow ub_k^j \times v_k \geq v_j \geq lb_k^j \times v_k \Rightarrow \frac{ub_k^j}{k_d} \times v_k \geq \frac{v_j}{k_d} \geq \frac{lb_k^j}{k_d} \times v_k$. Following Lemma 1, the theorem is proved. □

Theorem 3 *For a dependence vector $\vec{v}$, if $v_k \leq 0$, then*

$$\lceil \frac{-lb_k^j}{k_d} \rceil \times (-v_k) \geq \lceil \frac{v_j}{k_d} \rceil \geq \frac{v_j}{k_d} \geq \lfloor \frac{v_j}{k_d} \rfloor \geq \lfloor \frac{-ub_k^j}{k_d} \rfloor \times (-v_k)$$

where k_d is a positive integer.

Proof: For every dependence vector $\vec{v}$, $ub_k^j \geq \frac{v_j}{v_k} \geq lb_k^j$ and $v_k \leq 0 \Rightarrow lb_k^j \times v_k \geq v_j \geq ub_k^j \times v_k \Rightarrow \frac{-lb_k^j}{k_d} \times (-v_k) \geq \frac{v_j}{k_d} \geq \frac{-ub_k^j}{k_d} \times (-v_k)$. Following Lemma 1, the theorem is proved. □

For our 2-dimensional example, it is easier to understand the uniformization with the following corollaries by substituting $k_d = 1, j = 1$ and $k = 2$ in the above theorems.

Corollary 1 *For any valid dependence vector $\vec{v} = (v_1, v_2)$ in a 2-dimensional iteration space, if $v_2 \geq 0$, then $\vec{v} = (v_2 \times \lfloor lb_2^1 \rfloor + r, v_2) = v_2 \times (\lfloor lb_2^1 \rfloor, 1) + r \times (1, 0)$ where $r = v_1 - v_2 \times \lfloor lb_2^1 \rfloor$ and $r \in \mathbf{Z}^+ \cup \{0\}$.*

Corollary 2 *For any valid dependence vector $\vec{v} = (v_1, v_2)$ in a 2-dimensional iteration space, if $v_2 \leq 0$, then $\vec{v} = (-v_2 \times \lfloor -ub_2^1 \rfloor + r, v_2) = (-v_2) \times (\lfloor -ub_2^1 \rfloor, -1) + r \times (1, 0)$ where $r = v_1 - v_2 \times \lfloor -ub_2^1 \rfloor$ and $r \in \mathbf{Z}^+ \cup \{0\}$.*

The above two corollaries can be applied to the loop in Figure 2. For the dependence distance vectors (d_1, d_2) with $d_2 > 0$, we use Corollary 1 and a lower bound of $\frac{d_1}{d_2}$ is 5.16. For the dependence distance vectors with $d_2 < 0$, we use

Corollary 2 and an upper bound of $\frac{d_1}{d_2}$ is -2.06. Therefore, we found a feasible *BDVS*=$\{(1,0),(\lfloor 5.16 \rfloor,1),(\lfloor 2.06 \rfloor,-1)\} = \{(1,0),(5,1),(2,-1)\}$, which is the one shown in Example 5 of the previous section. The idea behind this 2-dimensional uniformization scheme is to find lb_2^1 (ub_2^1) for valid dependences (d_1, d_2) with $d_2 > 0$ ($d_2 < 0$). Then $(1,0)$ and $(\lfloor lb_2^1 \rfloor, 1)$ can be used to uniformize all dependences with $d_2 > 0$. Similarly, $(1,0)$ and $(\lfloor -ub_2^1 \rfloor, -1)$ can be used to uniformize all dependences with $d_2 < 0$. The advantage of this scheme is that the resulting dependence cone will be close to that of the original dependence cone of the non-uniform dependences. In fact, if all valid dependences have $\mid d_1 \mid \geq \mid d_2 \mid$, the dependence cone size will not be greater than $\frac{\pi}{2}$, whereas schemes such as that used in Example 4 have a dependence cone size of at least $\frac{3\pi}{4}$.

The two bounds needed, lb_2^1 and ub_1^2, can be computed without separating the dependences into the two groups as described above, which requires the consideration of a dependence direction vector with Eq. (6). It is not difficult to see that if $ub_1^2 \leq 0$, then there are no valid dependences with $d_2 > 0$ in the loop. Otherwise, we can use $\frac{1}{ub_1^2}$ as lb_2^1. Similarly, we can use $\frac{1}{lb_1^2}$ to compute ub_2^1. Finally, because both lb_1^2 and ub_1^2 consider all dependences and Eq. (6) gives the same ratio $\frac{d_2}{d_1}$ regardless of the dependence direction vectors, we avoid the complexity in considering the direction vectors as required by other schemes (Examples 2 and 3). This also means that we can use the *DCH* shown in Eq. (5) without further consideration of dependence directions. Besides, it works well with multiple dependences, in which case we only need to find the maximal upper and the minimal lower bounds of the ratios for all dependences.

The remaining question is how to obtain upper and lower bounds. Also, we find that the tighter the bounds, the smaller the dependence cone size we get. The best choice of the bounds are, of course, the maximum and the minimum. In the Appendix, we show that given Eq. (6), the maximal and the minimal ratios $\frac{d_i(T)}{d_1(T)}$ occur at the vertices of the *DCH* if $d_1(T)$ is never 0 in the *DCH*. Another issue is the generalization to a higher dimensional iteration space. Although our example is only for the 2-dimensional case, Theorems 2 and 3 are not restricted to the 2-dimensional cases. This issue is discussed in more detail in Section 6.

3.3 Static Strip Scheduling

The second step of the uniformization process is to find a loop scheduling scheme that can fully exploit the parallelism. We propose a *Static Strip Scheduling* scheme which can generally reduce communication overhead while improving the parallelism. Let $L_2(U_2)$ be the lower(upper) limit of the inner loop, $p < U_2 - L_2 + 1$ be the number of processors, and

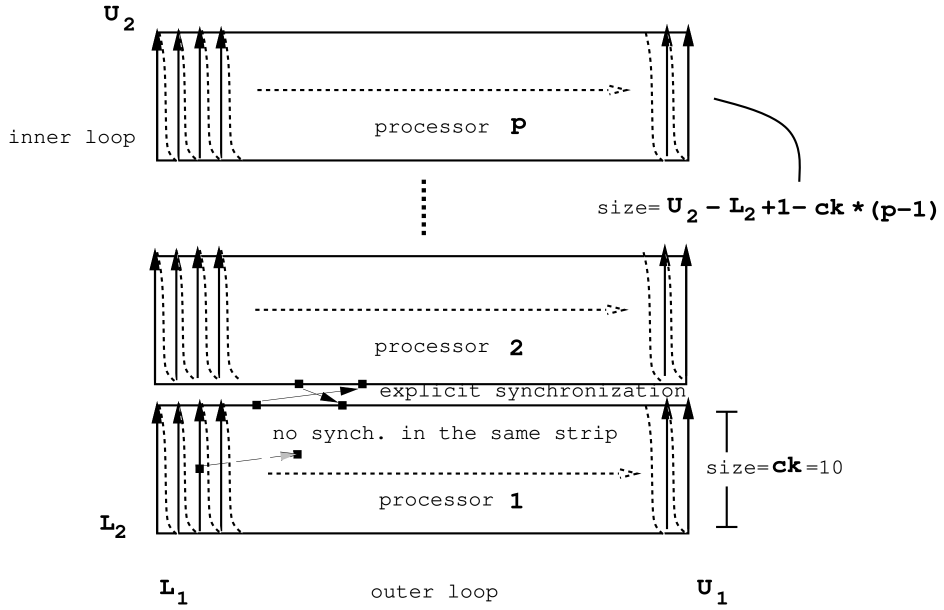

Figure 7: Static Strip Scheduling example.

let the chunk size be $ck = \lceil \frac{U_2 - L_2 + 1}{p} \rceil$. Suppose a *BDVS* has been selected to satisfy all of the dependence vectors in the loop. The Static Strip Scheduling scheme assigns the first ck iterations of the inner loop to the first processor, the second ck iterations to the second processor, and so on, as shown in Figure 7. This scheduling scheme takes advantage of the fact that all valid dependence vectors, (x, y), in the proposed *BDVS* (e.g., Example 5) always consist of $(1, 0)$, and other vectors in the *BDVS* always have offset $+1$ or -1 in the components other than the x-dimension. The dependence vector $(1, 0)$ can be automatically preserved by the forward execution of the outer loop in each processor. Other vectors in the *BDVS* have to be enforced only at the boundaries of the adjacent strips with explicit synchronization using synchronization primitives such as *post&wait* [20], *FULL/EMPTY bit* [24], *Cedar synchronization* [12], or by a process-based synchronization scheme as described in [25]. That is, many explicit synchronizations can be eliminated if the source and the sink of a dependence are in the same strip and are executed by the same processor. For the example given in Figure 7, suppose that $ck = 10$ and the vector $(5, 1) \in BDVS$. We must make sure the iteration $(i_1 = 3, i_2 = 5)$ is executed before $(i_1 = 8, i_2 = 6)$. The synchronization is not necessary (shown as a dashed arrow in the bottom strip of Figure 7) since the source and the sink are in the same strip, and the order is guaranteed by the sequential execution of the iterations in the strip. On the other hand, for the same dependence vector $(5, 1)$ at the boundaries (shown as solid arrows) of the strips, explicit synchronization is needed.

3.4 Summary

Following corollaries 1 and 2, we can construct *BDVS* similar to that in the Example 5 for all doubly-nested loops. However, it is possible that both $(0, 1)$ and $(0, -1)$ have to be included to cover all non-uniform dependences. It occurs when $ub_1^2 > 1$ and $lb_1^2 < -1$. Because, for example, if $ub_1^2 \not> 1$, $(1, 1)$ and $(1, 0)$ can be used to preserve all the dependence vectors with positive slopes, and these two *BDVS* vectors will not conflict with $(0, -1)$ even if it is needed for dependence vectors with slopes < -1. On the other hand, if $ub_1^2 > 1$ and $lb_1^2 < -1$, we need $(0, 1)$ for dependence vectors with slope > 1 and $(0, -1)$ for dependence vectors with slope < -1 and we have to solve the compatibility problem. We can use the *BDVS* proposed in [27] which is shown in Example 4. An *Index Synchronization* scheme was also proposed in [27] to parallelize the loop nests given a *BDVS* similar to that of Example 4. Figure 8 illustrates the index synchronization. The performance analysis for static strip scheduling and index synchronization is shown in a later section. We use the following *BDVS* selection algorithm for doubly-nested loops to summarize this section.

Algorithm 1
Begin

1. BDVS $\leftarrow \{(1, 0)\}$;
2. $\forall T \in$ DCH, *compute* ub_1^2 *and* lb_1^2, *lower and upper bounds of* $\frac{d_2(T)}{d_1(T)}$;
3. *if* ($ub_1^2 \leq 0$) *then do nothing;*
   else begin
   $lb_2^1 \leftarrow \frac{1}{ub_1^2}$;
   BDVS $\leftarrow$ BDVS $\cup \left\{(\lfloor lb_2^1 \rfloor, 1)\right\}$;
   end endif;
4. *if* ($lb_1^2 \geq 0$) *then do nothing;*
   else begin
   $ub_2^1 \leftarrow \frac{1}{lb_1^2}$;
   BDVS $\leftarrow$ BDVS $\cup \left\{(\lfloor -ub_2^1 \rfloor, -1)\right\}$;
   end endif;
5. *if both* $(0, 1)$ *and* $(0, -1)$ *are in the* BDVS *then*
   if ($| \lfloor lb_1^2 \rfloor | > | \lceil ub_1^2 \rceil |$ *and* $ub_1^2 \neq \infty$) *then* BDVS $\leftarrow \left\{(1, \lceil ub_1^2 \rceil), (0, -1)\right\}$;
   else BDVS $\leftarrow \left\{(1, \lfloor lb_1^2 \rfloor), (0, 1)\right\}$;

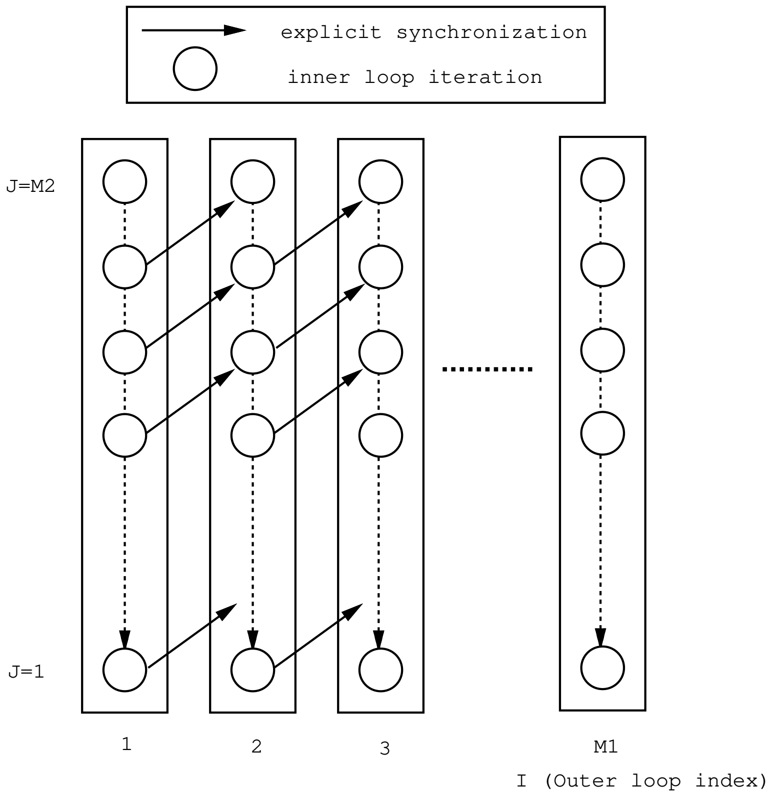

Figure 8: Index Synchronization for *BDVS* of Example 4.

endif;

End.

4 Performance Analysis for Static Strip Scheduling

We focus on two commonly discussed issues in parallel processing – parallelism and synchronization. In the following discussion, let p be the number of processors used, S be the amount of time taken for each inner loop iteration, and M_1, M_2 be the numbers of iterations in the outer and the inner loops, respectively. To simplify the discussion, we consider only the doubly-nested loops and assume p divides M_2. So the chunk size $ck = \lceil \frac{M_2}{p} \rceil = \frac{M_2}{p}$. Each strip in Figure 7 will have $ck \times M_1$ iterations, which is also the amount of work each processor is responsible for. The following formula is our model to predict the performance of the synchronization scheme,

$$\frac{M_2 \times M_1 \times S}{ck \times M_1 \times S + Z + O} \leq p \tag{11}$$

where $ck \times M_1 \times S$ is the useful work for each processor. The *dependence delay*, Z, is the amount of overall effective delay accumulated on the critical path. That is, Z is the total delay caused by dependence relations, and it determines the total parallelism even when other overheads are zero. The *communication overhead*, O, includes the time spent in scheduling and synchronization. Given M_1, M_2 and S, we want to have a small Z and a small O.

4.1 Parallelism

The value of the total delay Z depends largely upon the members of the *BDVS*, which in turn are determined by lb_2^1 and ub_2^1, the bounds of $\frac{d_1}{d_2}$, and whether the compatibility problem occurs. There are three cases after running Algorithm 1.

- (*Case 1*) If neither $(0, 1)$ nor $(0, -1)$ is in the *BDVS*, $Z_{case\ 1} = 0$ because all the vectors are *forward* in the sense that the sink has a larger outer loop index value than the source (see Example 5). If the progress of each strip stays more or less the same, the serial execution within each strip causes the sink to be executed after the source in most cases, although explicit synchronizations are still necessary. Because the processors never idle, $Z_{case\ 1} = 0$.

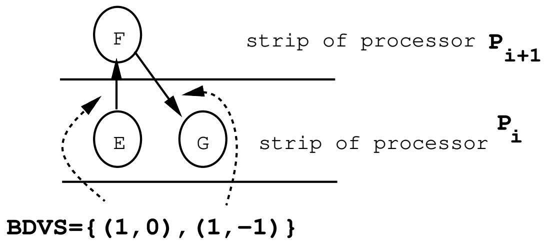

Figure 9: Worst scenario of *case 2*.

- (*Case 2*) If at the beginning of step 5 in Algorithm 1, one of the $(0, 1)$ and $(0, -1)$ is in the *BDVS*, then there is a delay between strips, because if $(0, 1)$ or $(0, -1) \in BDVS$, the initiation of each strip depends on the previous strip, hence causing the idle time in the processors. Specifically, for the first iteration of the outer loop, the execution of the inner loop iterations are serialized from top down or bottom up (see Figure 7). For the last processor, the delay time of the initiation is $ck \times (p-1) \times S$. After the initiation, if the progress of each strip is the same, no more delay is necessary. Therefore, $Z_{case\ 2} = ck \times (p-1) \times S = (M_2 - ck) \times S$. A special case occurs when $p > \frac{M_2}{2}$ and either $(1,1)$ or $(1,-1)$ is in the *BDVS* (see Figure 9). In this case, some strips have only one inner loop iteration for each outer loop iteration because $ck = \frac{M_2}{p} < 2$. The processor cannot execute consecutive iterations in such strip without having to wait for a source in the neighboring strip which waits for the previous iteration of the current strip. The critical path from E to G in Figure 9 is now $E \rightarrow F \rightarrow G$, and the length is $2 > ck = 1$. The delay Z in this special case is $((M_1 - 1) + ck \times (p-1)) \times S$. However, we usually will have $p \gg \frac{M_2}{2}$, and this worst case does not occur often.

- (*Case 3*) If both of the $(0,1)$ and $(0,-1)$ are in the *BDVS* at the beginning of the step 5 of Algorithm 1 and the compatibility problem occurs, then the index synchronization [27] as explained in the previous section has to be used. From Figure 8, we see the delay between consecutive outer loop iterations and the overall delay is $Z_{case\ 3} = (p-1) \times (1 + \mid \lceil ub_1^2 \rceil \mid) \times S$ or $Z_{case\ 3} = (p-1) \times (1 + \mid \lfloor lb_1^2 \rfloor \mid) \times S$.

4.2 Communication Overhead

Explicit synchronization is needed between tasks to ensure the correct result for parallel execution when there are dependences. This inter-processor communication takes time and could incur a lot of network contention, which in turn could cause further degradation in performance. With our static strip scheduling, a large portion of the synchronization within the same strip is contained in a single processor. The best example is the *BDVS* of *case 1* in the last section. Since the vectors have displacement of 1 in the y-dimension, only those iterations at the strip boundaries have to be synchronized with the adjacent processors. Given p processors, there are p strips and $p-1$ boundaries. Thus the number of synchronizations necessary for such a vector is $(M_1 - \lfloor lb_2^1 \rfloor) \times (p-1)$ or $(M_1 - \lfloor -ub_2^1 \rfloor) \times (p-1)$. In *case 2*, using the static strip scheduling, vector $(0,1)$ or $(0,-1)$ requires $M_1 \times (p-1)$ explicit synchronizations. In both cases, the number of synchronizations is fixed for a given p and M_1. So the more iterations the inner loop has, the smaller the percentage of the communication overhead will be. On the other hand, because processors are pre-assigned to the strips the scheduling overhead at the run-time is minimal. For *case 3*, almost every iteration requires explicit synchronizations (see Figure 8). The number of synchronizations is $(M_1 - 1) \times (M_2 - \mid \lceil ub_1^2 \rceil \mid)$ or $(M_1 - 1) \times (M_2 - \mid \lceil lb_1^2 \rceil \mid)$. If self-scheduling with a chunk size of 1 is used with index synchronization, there will be a total of M_1 schedulings performed.

	Cases	Parallelism	Synchronizations
1	$BDVS = \{(1,0),(5,1),(2,-1)\}$	p	$(2 \times M_1 - 5 - 2) \times (p-1)$
2	$BDVS = \{(1,0),(5,1),(0,-1)\}$	$\frac{M_1 \times M_2}{\frac{M_2}{p} \times M_1 + M_2 - \frac{M_2}{p}} = \frac{p}{1+\frac{p-1}{M_1}}$	$(2 \times M_1 - 5) \times (p-1)$
3	$BDVS = \{(1,1),(0,-1)\}$	$\frac{M_1 \times M_2}{\frac{M_2}{p} \times M_1 + (p-1) \times (1+1)} = \frac{p}{1+\frac{p-1}{M_1} \times \frac{2 \times p}{M_2}}$	$(M_1 - 1) \times (M_2 - 1)$

Table 1: Summary of performance analysis for static strip scheduling.

4.3 Summary

The discussion in the previous sections is summarized in Table 1. M_1 and M_2 are the number of iterations of the outer and the next outer loops, respectively, and we assume $p \leq \frac{M_2}{2}$ processors are used. We listed all three cases discussed so far. *Case 1* uses the *BDVS* of Example 5 generated by Algorithm 1. *Case 2* replaces $(\lfloor lb_2^1 \rfloor, 1)$ (or $(\lfloor -ub_2^1 \rfloor, -1)$) of the above *BDVS* with $(0,1)$ (or $(0,-1)$). *Case 3* uses *BDVS* of Example 4, which can be generated in the step 5 of Algorithm 1, and the index synchronization [27]. Note that all *BDVSs* of all three cases can be used to uniformize the dependences, but they have quite different performances. It is clear from the comparison below that *BDVS* of the *case 1* is the best since it provides the maximal parallelism and requires the least synchronization:

$$(2 \times M_1 - 5 - 2) \times (p-1) \leq (M_1 - 1) \times (2 \times p - 2) \leq (M_1 - 1) \times (M_2 - 2) < (M_1 - 1) \times (M_2 - 1)$$

The difference in the synchronization count is very large when $p \ll \frac{M_2}{2}$. *BDVS* of *Case 2* can be seen as a trade-off between the parallelism and the communication overhead. It provides slightly less parallelism than that of the *case 3* since $2 \times p \leq M_2$ (see the Parallelism column of Table 1). In return, it causes less synchronization on most occasions; that is, suppose $(0,-1)$ is in the *BDVS* and $lb_2^1 \geq 1$ (so that $(0,1)$ is not in the *BDVS*):

$$(2 \times M_1 - \lfloor lb_2^1 \rfloor) \times (p-1) \leq (2 \times M_1 - 1) \times (p-1) = (M_1 - \frac{1}{2}) \times (2 \times p - 2)$$

and is less than that of *case 3* ($(M_1 - 1) \times (M_2 - 1)$) if $p \ll \frac{M_2}{2}$ or $M_1 \geq \frac{M_2}{2}$ when $p = \frac{M_2}{2}$. It shows that even when either $(0,1)$ or $(0,-1)$ appears in the *BDVS* and the parallelism is not as high as expected with *case 1*, the reduction of the communication overhead may still make static strip scheduling an attractive choice.

To have a quantitative comparison, we let $M_1 = M_2 = 1000$ (that is, 1,000,000 total iterations) and $p = 10, 100, 500$. The amount of speedup and synchronization count are shown in Table 2 where we assume some constant values for O, the *communication overhead*, for Eq. (11) when calculating the speedup. For a smaller or moderate number of processors available, *BDVSs* of the *case 1* and *2* show the superority of requiring much less synchronization. Overall, *BDVS* of the *Case 1* provides higher efficiency through mostly “wait-free” synchronization.

Example	Schemes	Speedup			Synchronization
		overhead=0	overhead=5% $\times \frac{M_1 M_2}{p}$	overhead=10% $\times \frac{M_1 M_2}{p}$	count
$p = 10$	1	10	≈ 9.52	≈ 9.09	17937
	2	≈ 9.91	≈ 9.44	≈ 9.02	17955
	3	≈ 10	≈ 9.52	≈ 9.09	998001
$p = 100$	1	100	≈ 95.24	≈ 90.91	197307
	2	≈ 90.99	≈ 87.03	≈ 83.40	197505
	3	≈ 98.06	≈ 93.48	≈ 89.30	998001
$p = 500$	1	500	≈ 476.19	≈ 454.55	994507
	2	≈ 333.56	≈ 322.79	≈ 312.70	995505
	3	≈ 333.56	≈ 322.79	≈ 312.70	998001

Table 2: Performance examples ($M_1 = M_2 = 1000$).

5 Transformation Example

In this section, we show for our loop in Figure 2 how the final *BDVS* is determined and the loop parallelization transformation. Suppose we have a 50-processor multiprocessor system. We begin with the dependence whose source and sink are in statements S_p and S_q. We have solved the general form for dependence distance vectors as in Eq. (7). The 3-d *DCH* formed by the constraints in Eq. (5) has four vertices T_0, T_1, T_2, T_3 with their x, y, z coordinates as shown in Table 3. So $ub_1^2 = 0.19$ and $lb_1^2 = -0.48$ which result in $lb_2^1 = 5.16$ and $ub_2^1 = -2.06$. We now look at the output-dependences with

Point	x	y	z	d_1	d_2	$\frac{d_2}{d_1}$	$\frac{d_1}{d_2}$
T_0	285.14	144.57	285.14	714.86	138.57	0.19	5.16
T_1	8.00	6.00	8.00	22.00	0.00	0.00	$+\infty$
T_2	493.00	248.50	250.50	507.00	0.00	0.00	$+\infty$
T_3	331.33	6.00	331.33	668.67	-323.33	-0.48	-2.06

Table 3: Calculation of lb_2^1 and ub_2^1.

their source and sink both at statement S_q. For this kind of dependences, we have 4 variables and 2 equations. The derived general form for the dependence vector is $(d_1, d_2) = (0, 0)$; that is, there is no loop-carried output dependence from S_q to S_q. So the combined bounds for the dependence vectors are still the same. Next we use Algorithm 1 to generate the *Basic Dependence Vector Set* $BDVS = \{(1, 0), (5, 1), (2, -1)\}$ because there is no compatibility problem. Given 50 processors, the transformed loop is shown in Figure 10 after some simple optimizations to eliminate operations involving constants. We use a *semaphore* array s, which is allocated at each processor performing `P` operations and is initialized to all zeros. For ease of presentation, we assume it has 1000×1000 elements although we use it very sparsely. [3] The `P` and `V` operations are defined as

- `P(s): wait until s > 0 do s = s - 1`
- `V(s): s = s + 1.`

The optimal speedup can be analyzed as follows. For the first 20 iterations of the outer loop, only processor 1 is active. The total execution time for these 20 iterations is the time to execute the 210 iterations of the J-loop. Assuming that each iteration of the J-loop takes roughly the same time T_J to execute, that is, the computation in K-loop is small compared to the other part of the J-loop, then these 210 iterations take $210 \times T_J$ to complete. The rest of the 980 iterations of the I-loop will take $T_J \times 20 \times 980 = 19600 T_J$ to finish since each processor executes at most 20 J-iterations for each I-iteration. The total speedup is

$$\frac{T_J \times \textit{total number of iterations of J-loop}}{(210 + 19600)T_J} \approx 25.2$$

and the average processor utilization rate is about 50%. Nearly 90% of the dependence synchronization is saved by the static strip scheduling because many dependences do not cross the strip boundaries. The reason the parallelism is not 50 for 50 processors is that the iteration space for the doubly-nested loop is not rectangular, but is triangular. This makes the length of the strips for each processor different. Alternatively, it is possible to pursue higher speedup with more synchronizations if we use a smaller chunk size for each strip. For example, a scheme with 100 strips and a chunk size of 10 yields speedup of 33.6 for the same number of processors with twice as many synchronizations.

[3] In fact, if we use the *process-based* synchronization scheme as described in [25], which uses counter to inform other process the progress of the current process, we need only one counter per chunk. That is, the storage requirement is $O(p)$.

```
FOR EACH PROCESSOR P_pid DO
    DO I=1,1000,1
       L'_2 = 1 + (pid - 1) × 20
       U'_2 = min(pid × 20, I)
       IF (I ≥ 6 and pid > 1) P(s(I,L'_2))
       IF (L'_2 < U'_2 and pid > 1) V(s(I+2,L'_2-1))
       DO J=L'_2,U'_2,1
          IF (I ≥ 3 and J == pid × 20 and pid < 50) P(s(I,pid × 20))
          DO K=J,I
             ⋮
             ... =A(I+2K+5,4K-J)
          ENDDO
          ⋮
          A(I-J,I+J)= ...
          ⋮
          IF (J == L'_2 and pid > 1) V(s(I+2,L'_2-1))
       ENDDO
       IF (I ≥ 3 and U'_2 ≠ pid × 20 and pid < 50) P(s(I,pid × 20))
       IF (pid < 50) V(s(I+5,1 + pid × 20))
    ENDDO
ENDFOR
```

Figure 10: Transformed loop nest for Example 1

6 Discussion

Advantages

There are several properties that make the proposed *BDVS* and the static strip scheduling scheme attractive. First, the synchronization requires no atomic *read-modify-write* operation because all synchronization variables are modified by at most one processor. Second, the strips are assigned to the processors statically so the senders and the receivers of the synchronization signals are known at compile time. This is especially useful in *distributed shared memory* machines such as IBM RP3 [21], BBN Butterfly [5], and Stanford DASH [14], where a processor can access the local storage of other processors so each processor can busy-wait on the synchronization variables allocated locally but modified by remote processors. Such a technique reduces the busy-waiting traffic to the global communication network [26]. Also, synchronization will keep the execution of different processors more or less in lockstep which mitigates the load-balancing problem. Most important, static assignment of tasks to processors allows us to eliminate a lot of dependences within a strip.

Dependence Vector with One Free Variable

Sometimes we would encounter dependences whose general form of dependence vectors has only one free variable. In this case, it is possible to solve the only unknown free variable at run-time and find out the iteration containing the corresponding dependence source (or sink). The synchronization is applied only when there is an integer solution to the single integer free variable.

Extend Static Strip Scheduling to Higher Dimension

As we have seen in Section 4 and previous discussion, the proposed *BDVS* and static strip scheduling offers better speedup and less communication overhead if it can be used to uniformize the non-uniform dependences. So far, we have exploited parallelism from the two outermost perfectly nested loops. When the nesting level for the outermost perfectly nested loops is larger than 2, it is desirable to extend our scheme to exploit parallelism from more loop levels. We will briefly describe how this is done and discuss the extra overhead involved.

Suppose the iteration space is n-dimensional, $n \geq 2$. Similar to the 2-dimensional case, we need to find for each dimension $k = 2, \ldots, n$, a lower bound lb_j^1 as well as an upper bound ub_j^1. This can be accomplished using theorems in the Appendix to find ub_1^j and lb_1^j (because $lb_j^1 = \frac{1}{ub_1^j}$ and $ub_j^1 = \frac{1}{lb_1^j}$) once the general solution of the dependence distance vectors (e.g., Eq. (6)) is known. Then the *BDVS* equivalent to that generated from the first four steps of Algorithm 1 is

$$\left\{(1,0,\ldots,0),(\lfloor\frac{lb_2^1}{n-1}\rfloor,1,0,\ldots,0),(\lfloor\frac{-ub_2^1}{n-1}\rfloor,-1,0,\ldots,0),\ldots,(\lfloor\frac{lb_n^1}{n-1}\rfloor,0,\ldots,0,1),(\lfloor\frac{-ub_n^1}{n-1}\rfloor,0,\ldots,0,-1)\right\}.$$

We need to show that each dependence vector $\vec{v} = (v_1, \ldots, v_n)$, can be decomposed as a non-negative linear combination of the members of the above *BDVS*. Let $\vec{v} = \sum_{i=1}^{n} \lambda_i \vec{u_i}$ where $\vec{u_i}$ are members of the above *BDVS*. There are two cases:

1. For any $i \in \{2, \ldots, n\}$, if $v_i \geq 0$, then $\vec{u_i} = (\lfloor\frac{lb_i^1}{n-1}\rfloor, 0, \ldots, 1, \ldots, 0)$ and $\lambda_i = v_i$. Otherwise, $v_i \leq 0$, then $\vec{u_i} = (\lfloor\frac{-ub_n^1}{n-1}\rfloor, 0, \ldots, -1, \ldots, 0)$ and $\lambda_i = -v_i$.

2. Clearly with $n-1 > 0$, $v_1 \geq (n-1) \times \lfloor\frac{v_1}{n-1}\rfloor$. In addition, using $k_d = n-1, j = 1, k = i$ in Theorem 2 and 3, we obtain $\forall i = 2, \ldots, n$ $\lfloor\frac{v_1}{n-1}\rfloor \geq \lfloor\frac{lb_i^1}{n-1}\rfloor \times v_i = \lambda_i u_{i,1}$ (or $\lfloor\frac{v_1}{n-1}\rfloor \geq \lfloor\frac{-ub_n^1}{n-1}\rfloor \times (-v_i) = \lambda_i u_{i,1}$) where λ_i and $\vec{u_i}$ are determined in case 1 above $\Rightarrow v_1 = \sum_{i=2}^{n} \lambda_i u_{i,1} + r, r \geq 0$. Hence $\lambda_1 = r$.

We use an example in FIgure 11 to illustrate the *BDVS* needed to parallelize a triply-nested loop with non-uniform dependences. In this example, the matrix form of the dependence vectors is shown in Eq. (12).

```
        DO I=1,1001
           DO J=1,1001
              DO K=1,1001
                 ⋮
Sp:              ... =A(I,J+5,K+5)
                 ⋮
Sq:              A(I-J-K,J,K)= ...
                 ⋮
              ENDDO
           ENDDO
        ENDDO
```

Dep. source	Dep. sink	Dep. distance
...		
S_p(1,4,2)	S_q(17,9,7)	(16,5,5)
S_p(1,5,1)	S_q(17,10,6)	(16,5,5)
S_p(6,1,1)	S_q(18,6,6)	(12,5,5)
S_p(5,1,2)	S_q(18,6,7)	(13,5,5)
S_p(4,1,3)	S_q(18,6,8)	(14,5,5)
S_p(3,1,4)	S_q(18,6,9)	(15,5,5)
S_p(2,1,5)	S_q(18,6,10)	(16,5,5)
S_p(1,1,6)	S_q(18,6,11)	(17,5,5)
S_p(5,2,1)	S_q(18,7,6)	(13,5,5)
...		

Figure 11: A triply-nested loop with non-uniform dependence distance vectors.

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} + \begin{bmatrix} 10 \\ 5 \\ 5 \end{bmatrix} \tag{12}$$

The vertices of the 3-d *DCH* formed using constraints from the loop bounds are listed in Table 4 with values of $\frac{d_1}{d_2}$ and $\frac{d_1}{d_3}$ calculated. Consequently, we need a *BDVS*=

$$\left\{(1,0,0),(\lfloor\frac{2.40}{3-1}\rfloor,1,0),(\lfloor\frac{2.40}{3-1}\rfloor,0,1)\right\} = \{(1,0,0),(1,1,0),(1,0,1)\}$$

Point	x	y	z	d_1	d_2	d_3	$\frac{d_1}{d_2}$	$\frac{d_1}{d_3}$
T_0	1.00	1.00	1.00	12.00	5.00	5.00	2.40	2.40
T_1	989.00	1.00	1.00	12.00	5.00	5.00	2.40	2.40
T_2	1.00	989.00	1.00	1000.00	5.00	5.00	200.00	200.00
T_3	1.00	1.00	989.00	1000.00	5.00	5.00	200.00	200.00

Table 4: Calculation of $\frac{d_1}{d_2}$ and $\frac{d_1}{d_3}$.

to preserve all the original dependences. The reason that we use only 3 instead of $2n - 1 = 5$ vectors in the *BDVS* is because both of the lb_1^2 and lb_1^3 are greater than zero which eliminates the need of the vectors with -1 at the second or third dimenstions. The scheduling method is analogous to that of the 2-dimensional case. For example, in a 3-dimensional iteration space, we have "3-dimensional bars" instead of strips (see Figure 12). The factor of saving in communication

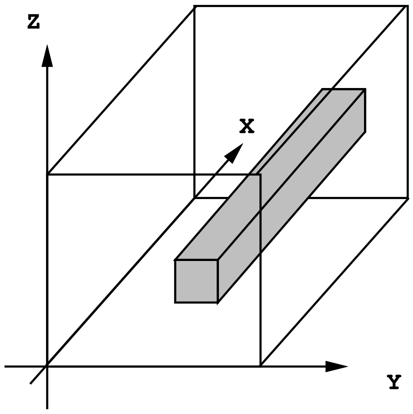

Figure 12: Static strip scheduling in a 3 dimensional iteration space.

overhead is similar to the volume-to-surface ratio of the 3-dimensional bars. Note that because we have a larger *BDVS* (cardinality $\leq 2n-1$), the extra communication overhead compared to a 2-dimensional scheme should be considered with the extra parallelism. This assumes that there is no compatibility problem. If in any dimension, both of the $(0, 0, \ldots, 1, \ldots, 0)$ and $(0, 0, \ldots, -1, \ldots, 0)$ are in the *BDVS*, a new *BDVS* will be used as in the step 5 of the Algorithm 1.

Dependence Distance Vector Optimization

In order to create *case 1 BDVS*, which provides better parallelism with less communication overhead, we want to have $\forall i = 2, \ldots, n$, $\lfloor lb_i^1 \rfloor \geq 1$ and $\lfloor -ub_i^1 \rfloor \geq 1$. In general, that means we want to have dependence distance vectors to have a larger displacement at the outermost loop level. Here we describe two methods to achieve this goal. It is called the *Dependence Distance Vector Optimization* and deserves further study.

If some of the entries in the dependence distance vectors Eq. (6) are zeros, they correspond to *DOALL* loops and can be interchanged freely with other loop nests without violating dependence relations [29]. In parallelization, it is desirable to have *DOALL* loops at the outer levels so that the granularity of the tasks can be increased to amortize the overhead.

Therefore, we should try to move all *DOALL* loops to the outermost levels. If these *DOALL* loops provide large enough parallelism, we need not consider *DOACROSS* parallelism further. Otherwise, we could coalesce those loops with the outermost *DOACROSS* loop. The net effect is that the component of the dependence distance vector corresponding to the coalesced *DOACROSS* loop will increase by a factor equal to the number of the total iterations of the *DOALL* loops. This can greatly reduce the chance of having $\lfloor lb_i^1 \rfloor$(or $\lfloor -ub_i^1 \rfloor$)$= 0$ to appear in the first component of the members in the *BDVS*. Hence, its parallelization performance is the best as has been discussed in *case 1* of Section 4, although the synchronization overhead might increase because we have more iterations in the outer loop now (see M_1 of the Synchronization column in Table 1).

More recently, a unified framework for loop transformation, called *Unimodular Transformation*, has become increasingly popular [4, 28]. Given more accurate characterization on the range of each entry in the dependence distance vectors (Eq. (6)), we can determine when loop reversal, interchange, or skewing is legal. By considering these transformations, more loops will fall into the category where *BDVS* of *case 1* can be constructed by the Algorithm 1. For example, if all dependence vectors of a doubly-nested loops are in the *region IV* of Figure 13, a reversal of the direction of the inner loop followed by a

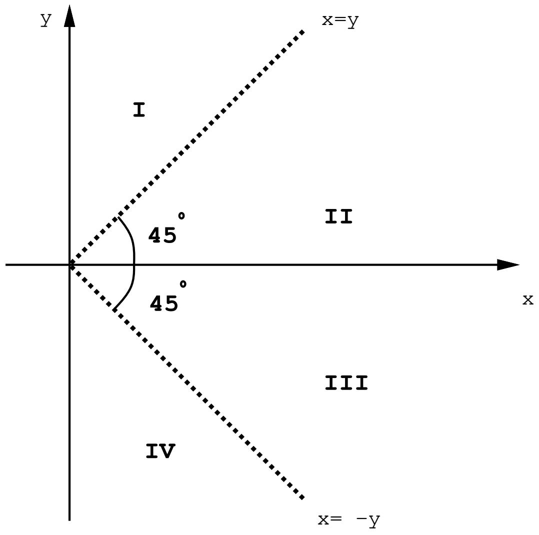

Figure 13: Regions of the dependence distance vector space.

loop interchange can put all dependence vectors into *region II* of the new iteration space. *BDVS* of *case 1* can be obtained. The static strip scheduling scheme can then be applied.

7 Previous Work

Tzen and Ni [27] proposed the first *Dependence Uniformization* method to parallelize doubly-nested loops with non-uniform affine dependences. They defined the notion of *Dependence Slope* which is the ratio of d_1 and d_2 for a dependence distance vector (d_1, d_2) in a doubly-nested loops. A pair of upper and lower bounds for the dependence slope, sl_{max} and sl_{min}, are then found at the vertices of the *DCH*. It is shown that either $\{(1, \lceil sl_{max} \rceil), (0, -1)\}$ or $\{(1, \lfloor sl_{min} \rfloor), (0, 1)\}$ can cover all possible non-uniform dependences. An *Index Synchronization* scheme was described to enforce these two sets of dependence vectors.

Their work can also be analyzed with Theorem 2 if we let $j = 2, k = 1$, and $k_d = 1$. Since $v_1 \geq 0$ for all valid dependence vectors, it implies that $v_2 = v_1 \times \lceil ub_1^2 \rceil - r, r \in \mathbf{Z}^+ \cup \{0\}$. That is, $(v_1, v_2) = v_1 \times (1, \lceil ub_1^2 \rceil) + r \times (0, -1)$ which is the same result as described above. One key difference is that their method will find one *BDVS* for all dependences in the loop while we treat different sets of non-uniform dependences differently to find the best *BDVSs* (see steps 3 and 4 of Algorithm 1). Only when these *BDVSs* are not compatible do we find bounds for *all* dependences (see step 5 of Algorithm 1). Example 4 is a result of applying their algorithm. However, because one of the "vertical" vectors $((0, 1), (0, -1))$ has to be in their *BDVS*, the dependence cone size will remain large and the parallelism suffers. In addition, the index synchronization requires explicit synchronization for every iteration, which will increase proportionally to the total number of loop iterations.

In one of our earlier papers [8], we described our scheme focusing on doubly-nested loops. More recent work have led us to a new framework as described in Theorems 2 and 3. It can also easily be extended to a higher dimensional iteration space.

Chen and Shang [9] discussed several optimization metrics for affine dependence uniformization such as schedule lengths, *BDVS* cardinalities and dependence cone sizes. They also described two basic methods for generating initial *BDVSs* (see Example 2 and 3) and then showed how to optimize these initial *BDVSs*. Their work provided valuable insights to the problem. But the *BDVS* found in their basic methods do not consider dependence directions and one of them cannot incorporate more than one dependence easily. Furthermore, the optimization for the dependence cone sizes, which greatly affects the overall parallelism, was not studied.

8 Conclusion

We presented a useful scheme to parallelize *DOACROSS* loops with non-uniform dependence distance vectors. This scheme can handle general loop nests such as non-perfectly nested loops as well as *triangular* or *trapezoidal* loops. The scheme takes into account the synchronization and the scheduling overhead in order to achieve high processor utilization. Our analytical results showed promising speedup and significantly less synchronization overhead than previously proposed schemes.

References

[1] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *ACM Symp. on Principles of Programming Languages*, pages 63–76, Jan. 1987.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capability. *AFIPS Spring Joint Computer Conf.*, 30:483–487, August 1967.

[3] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, 1988.

[4] U. Banerjee. Unimodular transformation of double loops. In *3rd Workshop on Programming Languages and Compilers for Parallel Computing*, August 1990.

[5] BBN Advanced Computers. *Butterfly products overview*, 1987.

[6] D.-K. Chen, J. Torrellas, and P.-C. Yew. An efficient algorithm for the run-time parallelization of DOACROSS loops. In *Supercomputing '94*, pages 518–527, November 1994. Also available as CSRD tech report No. 1345.

[7] D.-K. Chen and P.-C. Yew. An empirical study of DOACROSS loops. In *Supercomputing '91*, pages 620–632. IEEE Computer Society Press, November 1991. Also available as CSRD tech report No. 1140.

[8] D.-K. Chen and P.-C. Yew. A scheme for effective execution of irregular DOACROSS loops. In *Int'l. Conf. on Parallel Processing*, pages 285–292, August 1992. Also available as CSRD tech report No. 1192.

[9] Z. Chen and W. Shang. On uniformization of affine dependence algorithms. In *4th IEEE Conference on Parallel and Distributed Processing*, December 1992.

[10] R. Cytron. *Compile-time Scheduling and Optimization for Asychronous Machines*. PhD thesis, University of Illinois at Urbana-Champaign, 1984.

[11] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Int'l. Conf. on Parallel Processing*, pages 836–845, August 1986.

[12] P. Emrath, D. Padua, and P.-C. Yew. Cedar architecture and its software. In *Hawaii Int'l Conf. on System Science*, January 1989.

[13] L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, Feb. 1974.

[14] D. Lenoski, K. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Int'l. Symp. on Computer Architecture*, pages 148–159, May 1990.

[15] S. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Trans. on Computers*, C-36(12), December 1987.

[16] Y. Muroaka. *Parallelism Exposure and Exploitation in Programs*. PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Feb. 1971.

[17] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Comm. of ACM*, pages 1184–1201, Dec. 1986.

[18] D. A. Padua. *Multiprocessors: Discussion of Some Theoretical and Practical Problems*. PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, October 1979.

[19] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans. on Computers*, c-29(9):763–776, September 1980.

[20] The Parallel Computing Forum. *PCF Fortran: Language definition*, 1 edition, August 1988.

[21] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *1985 Int'l. Conf. on Parallel Processing*, pages 764–771, August 1985.

[22] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[23] Z. Shen, Z. Li, and P.-C. Yew. An empirical study of fortran programs for parallelizing compilers. *IEEE Trans. on Parallel and Distributed System*, 1(3):356–364, July 1990.

[24] B. J. Smith. A pipelined, shared resource mimd computer. In *Int'l. Conf. on Parallel Processing*, pages 6–8, Aug. 1978.

[25] H.-M. Su and P.-C. Yew. On data synchronization for multiprocessors. In *Int'l. Symp. on Computer Architecture*, pages 416–423, may 1989.

[26] H.-M. Su and P.-C. Yew. Efficient doacross execution for distributed shared memory multiprocessors. In *Supercomputing '91*, Nov. 1991.

[27] T. H. Tzen and L. Ni. Dependence uniformization: A loop parallelization technique. *IEEE Trans. on Parallel and Distributed Systems*, 4(5), May 1993.

[28] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, 1991.

[29] M. J. Wolfe. *Optimizing Compilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, 1982.

[30] C.-Q. Zhu and P.-C. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. on Software Enginering*, pages 726–739, June 1987.

A Dependence Convex Hull and Extreme Values of $\frac{d_j}{d_k}$

In this appendix, we show that the constraints from Eq. (2) form a convex polytope in $\mathbf{R^n}$ and show how to find the extreme values of $f(T) = \frac{d_j(T)}{d_k(T)}$ for T in that convex polytope. These extreme values can be used as upper and lower bounds (ub_k^j and lb_k^j) in Theorem 2 and 3.

Similar to the notation used in Section 2.3, we let $l_p + l_q$ be the number of loop indices used in the reference pair and m is the number of the linearly independent equations in Eq. (1). We assume $n = l_p + l_q - m$, $T = (t_1, \ldots, t_n) \in \mathbf{R^n}$, $d_k(T) = T\vec{a}^T + a_0$, and $d_j(T) = T\vec{b}^T + b_0$, where $\vec{a} = (a_1, \ldots, a_n)$ and $\vec{a}^T$ is the transposition of $\vec{a}$ in the following discussion.

Theorem 4 *The intersection* V *of the half-spaces, which is formed by substituting Eq. (4) into Eq. (2), is a convex polytope in* $\mathbf{R^n}$.

Proof: After substituting Eq. (4) into Eq. (2), the constraints in Eq. (2) are closed half-spaces in $\mathbf{R^n}$ and therefore are convex. The intersection *V* of these convex sets is also convex. Eq. (4), which is the general solution to Eq. (1) solved using the algorithm in [3], has the following matrix form

$$(i_1, \ldots, i_{l_p}, j_1, \ldots, j_{l_q}) = \mathbf{tG} \tag{13}$$

where $\mathbf{G}$ is an $(m+n) \times (m+n)$ *unimodular* matrix and $\mathbf{t} = (c_1, \ldots, c_m, t_1, \ldots, t_n)$. The first m components, $c_1, \ldots, c_m$, are integer constants while the last n components of $\mathbf{t}$ are free variables. The loop bounds for the $l_p + l_q$ loops imply that all the components of the left-hand side of Eq. (13) are bounded, so there exists two $1 \times (m+n)$ integer constant matrices $\mathbf{l}$ and $\mathbf{u}$ such that

$$\mathbf{l} \le \mathbf{tG} \le \mathbf{u}.$$

Since all the components in $\mathbf{l}$ and $\mathbf{u}$ are integer constants and $\mathbf{G}$ is *non-singular*, all the components of $\mathbf{t}$ are bounded. This implies that the intersection *V* is a closed and bounded convex polyhedral set and therefore it is a convex polytope in $\mathbf{R^n}$. □

Theorem 5 *The maximal (minimal) values of* $f(T) = \frac{d_j(T)}{d_k(T)}$ *in a convex polytope* V *of* $\mathbf{R^n}$ *occurs at the vertices of* V *if there are no* $T_1, T_2 \in$ V *such that* $d_k(T_1)d_k(T_2) < 0$ *(a proof for* $\mathbf{R^2}$ *is in [27]).*

Proof: Suppose on the contrary that the maximal value of $f(T)$ occurs at T_0 where T_0 is not a vertex of *V* and no vertex has a value of f greater than or equal to $f(T_0)$. We want to prove by contradiction that there is some point $T' \neq T_0$, $T' \in V$, such that $f(T') > f(T_0)$.

Case 1: T_0 is in the *relative interior* ($rel\_int(V)$) of *V*. Let the dimension of the *affine hull* (*aff*(*V*)) of *V* be n' and let the *aff*(*V*) be defined as $\{T : T\vec{h}_i^T = c_i, c_i \text{ is constant}, \forall i = 1, \ldots, n-n', T \in \mathbf{R^n}\}$. Our goal now is to find a vector $\vec{\Delta} = (\delta_1, \ldots, \delta_n)$ which is an *ascent direction* at T_0 in the *affine hull*. That is,

$$\vec{\Delta}(\nabla f(T_0))^T = \vec{\Delta}\left(\frac{d_k(T_0)\vec{b} - d_j(T_0)\vec{a}}{(d_k(T_0))^2}\right)^T > 0 \tag{14}$$

and

$$\vec{\Delta}\vec{h}_i^T = 0, \qquad \forall i = 1, \ldots, n-n'. \tag{15}$$

First we show that $d_k(T_0) \neq 0$. Suppose $d_k(T_0) = 0$, there are two possibilities. If *V* is entirely in the hyperplane $d_k(T) = 0$, then all points, including the vertices, will have the same value for f and the theorem is correct. Otherwise, the hyperplane $d_k(T) = 0$ will divide *V* into three non-empty sets

$$\begin{aligned} H^- &= \{T : d_k(T) < 0, T \in V\} \\ H^0 &= \{T : d_k(T) = 0, T \in V\} \\ H^+ &= \{T : d_k(T) > 0, T \in V\}, \end{aligned}$$

which clearly contradicts the assumption that there are no $T_1, T_2 \in V$ such that $d_k(T_1)d_k(T_2) < 0$. So we only need to discuss the situation when $d_k(T_0) \neq 0$.

Next, we want to show that neither $d_k(T_0)\vec{b} - d_j(T_0)\vec{a}$ equals $\vec{0}$ nor it is linearly dependent of $\{\vec{h_i}\}$. If $d_k(T_0)\vec{b} - d_j(T_0)\vec{a} = \vec{0}$, then $f(T_0) \times \vec{a} = \vec{b}$ and

$$f(T_0) = \frac{d_j(T_0)}{d_k(T_0)} = \frac{f(T_0) \cdot T_0\vec{a}^T + b_0}{T_0\vec{a}^T + a_0} \Rightarrow b_0 = f(T_0) \times a_0 \Rightarrow f(T) = \frac{f(T_0) \cdot T\vec{a}^T + f(T_0) \cdot a_0}{T\vec{a}^T + a_0} = f(T_0)$$

which implies that $f(T)$ is a constant for all points $T \in V$ including the vertices, a contradiction.

On the other hand, if $d_k(T_0)\vec{b} - d_j(T_0)\vec{a}$ is linearly dependent of the set of vectors $\{\vec{h_i}\}$, then there exists a set of real constants $\{e_i\}$ such that

$$d_k(T_0)\vec{b} - d_j(T_0)\vec{a} = \sum_{1}^{n-n\prime} e_i\vec{h_i}. \tag{16}$$

It implies that every $\vec{\Delta}$ which satisfies Eqn. (16) will have

$$\vec{\Delta}(d_k(T_0)\vec{b} - d_j(T_0)\vec{a})^T = 0 \Rightarrow \vec{\Delta}\vec{b}^T = f(T_0) \cdot \vec{\Delta}\vec{a}^T \Rightarrow f(T_0 + \vec{\Delta}) = \frac{f(T_0) \cdot d_k(T_0) + f(T_0) \cdot \vec{\Delta}\vec{a}^T}{d_k(T_0) + \vec{\Delta}\vec{a}^T} = f(T_0).$$

That means $f(T) = f(T_0)$ for all $T \in$ *aff*(V) and the vertices are included, a contradiction.

Therefore, if $n' = 0$, *aff*(V) is a point and the theorem is true. Otherwise, we show that an *ascent direction* at T_0 in the *affine hull* exists because $(n - n') < n$ and there exist $\vec{\Delta}$ which satisfy both Eq. (14) and Eq. (15). Since $T_0 \in rel\_int(V)$, there exists $\epsilon > 0$ such that $\{T' : \| T' - T_0 \| < \epsilon\} \cap$ *aff*$(V) \subseteq V$. So $T' = T_0 + \frac{\vec{\Delta}}{\|\vec{\Delta}\|} \times \frac{\epsilon}{2}$ will have $f(T') > f(T_0)$, a contradiction.

Case 2: If T_0 is on the boundary of V, then it is in the intersection of hyperplanes which bound V. Let $B = \{T : T\vec{h_i}^T = c_i, i = 1, \ldots, n - n', T \in V\}$ be such intersection with the least number of dimensions, n'. T_0 must be in the relative interior of B. We can use the similar approach above to show that unless $n' = 0$, in which case B contains a single vertex T_0, either $f(T)$ is constant for all $T \in B \cap V$ or there exists $T' \in B \cap V$ such that $f(T') > f(T_0)$, a contradiction.

Since both of the above cases contradict our assumption, we conclude that the maximal value of $f(T) = \frac{d_j(T)}{d_k(T)}$ must occur at one of the vertices of V. The same argument applies for the minimal values, too. □

Because $d_k(T) = 0$ is a hyperplane and V is convex, we can determine whether $d_k(T) = 0$ passes through V, or equivalently, whether there exist two points T_1 and T_2 such that $d_k(T_1)d_k(T_2) < 0$ by evaluating $d_k(T)$ at all vertices. If there exist two vertices T_1 and T_2 of V such that $d_k(T_1)d_k(T_2) < 0$, then $d_k(T) = 0$ passes through V. Otherwise $d_k(T) \geq 0$ (or ≤ 0) for all $T \in V$ and theorem 5 can be applied.

Finally, if the above theorem can not be used to derive a tighter bound for $f(T) = \frac{d_j(T)}{d_k(T)}$, we have to use the the most conservative bound $(-\infty, +\infty]$.

List of Figures

List of Tables